Anton Kolkma (nr. 2641122) & Maxim Abramov (nr. 2673452)
Concurrency & Multithreading (X_401031)
Vrije Universiteit Amsterdam

Programming Assignment Report

Introduction

Model checking algorithms are methods to check whether a finite-state model of a system violates certain temporal properties. In this assignment, we aimed to translate the nested depth-first search (NDFS) algorithm, which is inherently sequential, to an implementation in Java that makes use of multi-core architectures.
In this assignment, we have implemented Alg 2. of the paper making use of static (global) structures accessible to the worker class and local structures for the colors, unique to each worker's execution path. Additionally, we have moved the responsibility for setting up multiple workers into the NNDFS class, which creates them; this removes the necessity for indexing our workers. To implement different permutations of successors during the exploration of the graph, we randomly shuffle them for each worker, which results in a more diversity of paths, which means that most of the time different workers explore different paths in the graph.

Design

There are a couple of key differences between the original NDFS and the MC-NDFS algorithm as pointed out in the Laarman paper. The multi-core version now shares colors between the workers, namely states can be globally colored red. In our naive implementation, this is represented as a static hashmap, which functions as a global data structure that all workers can access. Another data structure that is introduced is a global count in each state. This is also a static hashmap in the worker class. Lastly, workers are able to locally set states to pink to signify states of the red DFSs, however since these are local they are simply a hashmap local to each worker.

Challenges in the naive implementation

There were certain challenges that needed to be addressed in order to translate the Laarman MC-NDFS algorithm into a Java implementation. Below is a list of challenges and how we solved them for the naive implementation:

1. What data needs to be shared, and what doesn't?
   Since the MC algorithm allows the sharing of the red states and the count in each state with other workers, these data structures must be shared. Since the other colors, including pink are local in nature, they are not shared with the other workers.
2. Which data structures are we going to use? What are the advantages and disadvantages of these?
   We decided to use a HashMap to keep track of states as the key with either a Color, Boolean, or Integer as the value. An advantage of this is that it simplifies the tracking of states by simply performing a lookup in the map. A disadvantage is that HashMaps cannot be concurrently operated on, thus it presents a problem for the global

HashMaps. This was solved naively by introducing synchronized blocks in sections of the algorithm where a thread tries to access these global data structures.

3. <u>How are we going to prevent threads from searching the same area of the graph?</u>
   In order to prevent threads from searching the same states, the list of states that the post-order function generates are shuffled. In this manner, it increases the probability that worker threads will search the graph from different locations.
4. <u>Based on the algorithm, how well do we expect our implementation to perform? What kinds of graphs would the algorithm perform well on, and which would it perform less so?</u>
   Based on the algorithm, we expect the multi-core implementation to perform better than the sequential version on graphs that contain accepting cycles. Due to the fact that there are multiple workers which traverse different areas of the graph introduced by the randomization of the post order, the probability that a worker finds a counter example is increased. However, on graphs which do not contain an accepting cycle, we do not expect a performance boost as each worker will have to fully traverse the state space.
5. <u>How will we terminate the search and our program?</u>
   In the MC algorithm, once a cycle has been found all workers must exit. In the Java implementation, after a cycle has been found, a shutDownNow() function is called which sends an interrupt signal to all the worker threads. Whilst the workers are performing the algorithm, they are checking for interrupts, and if it receives one, throws a CycleFoundException, which exits the algorithm. The main thread then waits for all workers to terminate, and returns whether or not a cycle has been found.

<u>The Improvement</u>

There are certain consequences in our solutions to the aforementioned challenges in the naïve implementation. One consequence that we identified was the use of HashMaps in representing the global data structures, namely the global counts and the global reds. In the naïve implementation, due to the fact that the code to increment, decrement, and get the value of the global count/reds are placed inside a synchronization block, there is an implicit lock that disallows other workers access to the critical section. This yields a performance overhead as when workers try to access the global count of different states, it would lock the whole data structure and force the workers to wait.

To improve the concurrency of these data structures, we changed the HashMap structure to a ConcurrentHashMap, which allows the worker threads to read elements from the map concurrently without locking. The performance of updating the map is also improved, as updating the value locks only a segment of the map, allowing other workers to read/update different keys (states). The change of this data structure also removes the need to place the code in a synchronized block, as the locking already happens in the data structure itself. However, removing synchronized blocks meant that modification of existing values was not atomic. Thus, we made use of AtomicIntegers instead of the unsafe Integer class, which allowed us to increment and decrement the values atomically and get rid of synchronization blocks, which were coarse-grained.

As a result, we managed to introduce the ability to access and modify the structure concurrently at a higher rate compared to the naïve version, increasing the performance whilst maintaining the correctness of the algorithm.

Evaluation

In general, we see that the MC-NDFS algorithm performs faster on graphs that contain accepting cycles. This validates our hypothesis since accepting cycles are found faster with a randomized walk, and, as the number of workers increases, the chances of finding a cycle increases as well.
For example, *bintree-converge* graph contains an accepting cycle, and our version manages to yield a performance speedup of almost 2x compared to the sequential one. Another example is *bintree-cycle-min* input, which the average runtime of finding the accepting cycle continuously decreases as the number of threads increases. This means that our MC-NDFS algorithm is scalable, at least on graphs with similar properties to *bintree-cycle-min*.

On the other hand, if a graph has no accepting cycles, additional workers only cause overhead when accessing shared data structures and decrease performance, such as the *bench-wide* graph. In this case we cannot determine that there are no accepting cycles in the graph until all of the workers report that they have found no cycles. This means, that we have to wait longer for them to finish execution, which results in more delays in the detection of a cycle. Additionally, using more workers on this type of graphs only increases set-up time and number of concurrent accesses to shared data structures, which decreases overall performance.

As we can see from the comparison table[fig.3], our improved version is, on average, about 10-20% faster than our naive implementation of the algorithm. In some specific cases, such as *bench-deep,* there is a significant performance boost when there are more than 8 threads. With 16 threads, it detects an accepting cycle in half the time of the naïve version on average. This is probably due to the fact that worker threads have more fine-grained concurrency capabilities as compared to the naïve version. On graphs which do not contain an accepting cycle, there does not seem to be a performance decrease, thus the improved version should be used instead of the naïve implementation as it yields an overall performance gain.

However, there is a trade-off between the naïve and the improved version: while the access to concurrent structures is more fine-grained, it requires concurrent classes, which might result in delays due to their sophisticated implementation.

Overall, we see that the performance speed ups that are gained/lost depends specifically on whether or not graphs have accepting cycles, as well as the characteristics of such graphs constitute an important role in whether or not the MC-NDFS algorithm scales well.

Average performance times of version 1[fig.1], version 2[fig.2] and a comparison[fig.3] between them are provided in the appendix.

Appendix

Below are tables showing average performance of each version compared to sequential implementation and to each other tested on 1, 2, 4, 8 and 16 threads.

“*” indicates that inputs have been run 20 times to stabilize the average runtime, due to randomness of the post order shuffling sometimes causing significant variances in runtime between multiple runs.
“**n/a**” indicates that execution on a DAS node takes too long to measure.

[fig.1] Average time in milliseconds over 10 executions with variable number of threads.

Input Graph	**Sequential version**	**Mcndfs_1_naive**				
		1	**2**	**4**	**8**	**16**
accept-cycle	0.9	5.1	5.7	6.0	8.8	16.2
bench-deep	4421.4	1264.8	1823.9	2727.5	7386.3	16085.0
bench-wide	22838.2	25702.8	25570.8	50665.0	107738.5	**n/a**
bintree-converge	3402.6	2582.0	1846.2	1834.7	1904.3	1903.1
bintree-cycle-max*	1.5	1436.8	1157.6	1223.5	1178.5	1555.5
bintree-cycle-min*	3117.8	1815.8	1225.8	1403.8	1329.3	1189.5
bintree-cycle	1.4	6.0	6.9	7.4	9.2	16.6
bintree-loop	3470.2	2690.1	1889.5	1799.9	1921.1	2018.4
bintree	3031.8	2494.8	1931.1	1844.2	1891.0	1956.0
simple-loop	0.8	5.0	6.5	6.6	9.6	15.2

[fig.2] Average time in milliseconds over 10 executions with variable number of threads.

Input Graph	**Sequential version**	**Mcndfs_2_improved**				
		1	**2**	**4**	**8**	**16**
accept-cycle	0.9	5.2	5.9	5.9	9.8	16.5
bench-deep	4421.4	1243.5	1745.6	2404.7	4524.0	8400.8
bench-wide	22838.2	25632.3	25934.6	49405.1	89356.9	**n/a**
bintree-converge	3402.6	2639.8	1902.4	1687.4	1525.5	1577.3
bintree-cycle-max*	1.5	1383.9	1231.3	1016.7	832.7	963.6
bintree-cycle-min*	3117.8	1377.5	1210.2	1090.1	888.5	789.9
bintree-cycle	1.4	5.9	6.9	7.3	9.2	17.2
bintree-loop	3470.2	2605.0	1972.6	1621.4	1613.2	1542.9
bintree	3031.8	2629.4	1883.9	1632.9	1538.8	1439.1
simple-loop	0.8	4.7	5.2	5.7	8.7	16.3

[fig.3] Comparison between the MC-NDFS naïve (1) and the improved versions (2), each input being run 10 times. Average time is given in milliseconds.

Input Graph	**Version**	**Number of Threads**				
		1	**2**	**4**	**8**	**16**
accept-cycle	1	5.1	5.7	6.0	8.8	16.2
	2	5.2	5.9	5.9	9.8	16.5
bench-deep	1	1264.8	1823.9	2727.5	7386.3	16085.0
	2	1243.5	1745.6	2404.7	4524.0	8400.8
bench-wide	1	25702.8	25570.8	50665.0	107738.5	**n/a**
	2	25632.3	25934.6	49405.1	89356.9	**n/a**
bintree-converge	1	2582.0	1846.2	1834.7	1904.3	1903.1
	2	2639.8	1902.4	1687.4	1525.5	1577.3
bintree-cycle-max*	1	1436.8	1157.6	1223.5	1178.5	1555.5
	2	1383.9	1231.3	1016.7	832.7	963.6
bintree-cycle-min*	1	1815.8	1225.8	1403.8	1329.3	1189.6
	2	1377.5	1210.2	1090.1	888.5	789.9
bintree-cycle	1	6.0	6.9	7.4	9.2	16.6
	2	5.9	6.9	7.3	9.2	17.2
bintree-loop	1	2690.1	1889.5	1799.9	1921.1	2018.4
	2	2605.0	1972.6	1621.4	1613.2	1542.9
bintree	1	2494.8	1931.1	1844.2	1891.0	1956.0
	2	2629.4	1883.9	1632.9	1538.8	1439.1
simple-loop	1	5.0	6.5	6.6	9.6	15.2
	2	4.7	5.2	5.7	8.7	16.3